EDGE

Python Programming and Basic Data Science

Lecture-2: NUMPY

Prof. Dr. G M Atiqur Rahaman
Computer Science and Engineering Discipline

Contents (Lec-4)
Introduction to NumPy

```
import numpy
numpy.__version__

import numpy as np
```

to display all the contents of the numpy namespace, you can type

this: `np.<TAB>`

to display NumPy's built-in documentation:

```
np?
```

Understanding Data Types in Python

```c
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```python
# Python code
result = 0
for i in range(100):
    result += i
```

Understanding Data Types in Python

Notice the main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred.

This means, for example, that we can assign any kind of data to any variable:

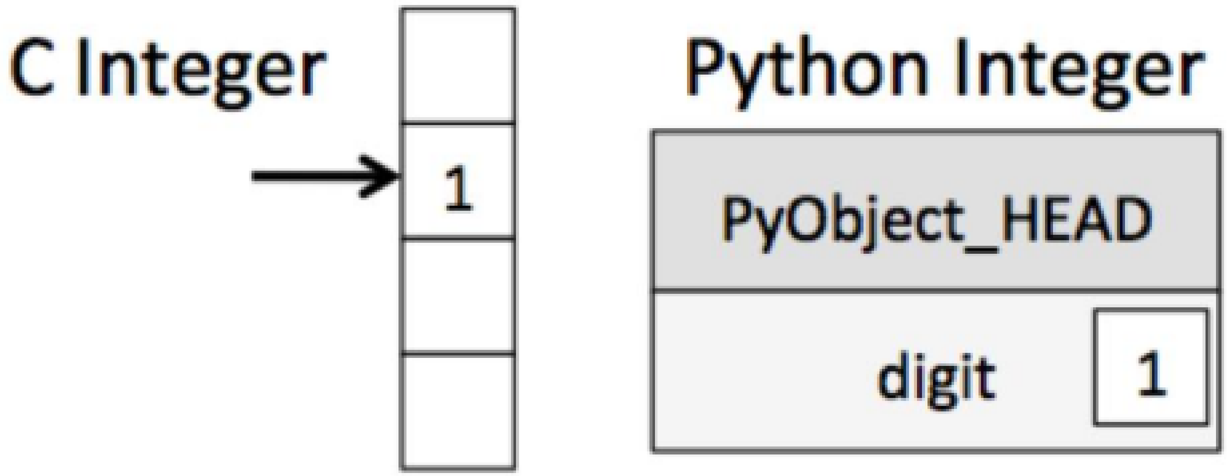
```
# Python code
x = 4
x = "four"
```

A Python Integer Is More Than Just an Integer

Looking through the Python 3.4 source code, we find that the integer (long) type definition effectively looks like this (once the C macros are expanded):

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

A Python Integer Is More Than Just an



Integer

Here PyObject_HEAD is the part of the structure containing:
the reference count,
type code, and
other pieces mentioned before.

A Python List Is More Than Just a List

The standard mutable multi-element container in Python is the list. We can create a list of integers as follows:

```
In [1]: L = list(range(10))
        L
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: type(L[0])
Out[2]: int
```

Or, similarly, a list of strings:

```
In [3]: L2 = [str(c) for c in L]
        L2
Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In [4]: type(L2[0])
Out[4]: str
```

A Python List Is More Than Just a List

Because of Python's dynamic typing, we can even create heterogeneous

```
In [5]: L3 = [True, "2", 3.0, 4]
        [type(item) for item in L3]

Out[5]: [bool, str, float, int]
```

lists:

A Python List Is More Than Just a List

In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in the following figure:

Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in array can be used to create dense arrays of a uniform
type:

```
In [6]: import array
        L = list(range(10))
        A = array.array('i', L)
        A

Out[6]: array('i', [0, 1, 2, 3,
```

Here 'i' is a type code indicating the contents are integers.

Much more useful, however, is the ndarray object of the NumPy package. While Python's array object provides efficient storage of array-based data, NumPy adds to this efficient operations on that data.

Creating Arrays from Python Lists

First, we can use np.array to create arrays from Python lists,

```
In [6]: import array
        L = list(range(10))
        A = array.array('i', L)
        A

Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

Creating Arrays from Python

Lists

```
In [9]: np.array([3.14, 4, 2, 3])
Out[9]: array([ 3.14,  4.  ,  2.  ,  3.  ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

```
In [10]: np.array([1, 2, 3, 4], dtype='float32')
Out[10]: array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensiona
here's one way of initializing a multidimensional array using a list of lists:

Creating Arrays from Python Lists

- Finally, unlike Python lists, NumPy arrays can explicitly be multi dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
In [11]: # nested lists result in multi-dimensional arrays
         np.array([range(i, i + 3) for i in [2, 4, 6]])

Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
In [12]: # Create a length-10 integer array filled with zeros
         np.zeros(10, dtype=int)

Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In [13]: # Create a 3x5 floating-point array filled with ones
         np.ones((3, 5), dtype=float)

Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])

In [14]: # Create a 3x5 array filled with 3.14
         np.full((3, 5), 3.14)

Out[14]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14]])

In [15]: # Create an array filled with a linear sequence
         # Starting at 0, ending at 20, stepping by 2
         # (this is similar to the built-in range() function)
         np.arange(0, 20, 2)

Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Creating Arrays from

Scratch

```
In [16]: # Create an array of five values evenly spaced between 0 and 1
         np.linspace(0, 1, 5)

Out[16]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

In [17]: # Create a 3x3 array of uniformly distributed
         # random values between 0 and 1
         np.random.random((3, 3))

Out[17]: array([[ 0.99844933,  0.52183819,  0.22421193],
                [ 0.08007488,  0.45429293,  0.20941444],
                [ 0.14360941,  0.96910973,  0.946117  ]])

In [18]: # Create a 3x3 array of normally distributed random values
         # with mean 0 and standard deviation 1
         np.random.normal(0, 1, (3, 3))

Out[18]: array([[ 1.51772646,  0.39614948, -0.10634696],
                [ 0.25671348,  0.00732722,  0.37783601],
                [ 0.68446945,  0.15926039, -0.70744073]])
```

Creating Arrays from Scratch

In [19]:
```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

Out[19]:
```
array([[2, 3, 4],
       [5, 7, 8],
       [0, 5, 0]])
```

In [20]:
```
# Create a 3x3 identity matrix
np.eye(3)
```

Out[20]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

In [21]:
```
# Create an uninitialized array of three integers
# The values will be whatever happens to already exist at that memory
np.empty(3)
```

Out[21]:
```
array([ 1.,  1.,  1.])
```

NumPy Standard Data Types

- NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.
- The standard NumPy data types are listed in the following table. Note that when constructing an array, they can be specified using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

NumPy Standard Data Types

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

The Basics of NumPy Arrays

We'll cover a few categories of basic array manipulations here:

- **Attributes of arrays**: Determining the size, shape, memory consumption, and data types of arrays
- **Indexing of arrays**: Getting and setting the value of individual array elements • **Slicing of arrays**: Getting and setting smaller subarrays within a larger array • **Reshaping of arrays**: Changing the shape of a given array
- **Joining and splitting of arrays**: Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

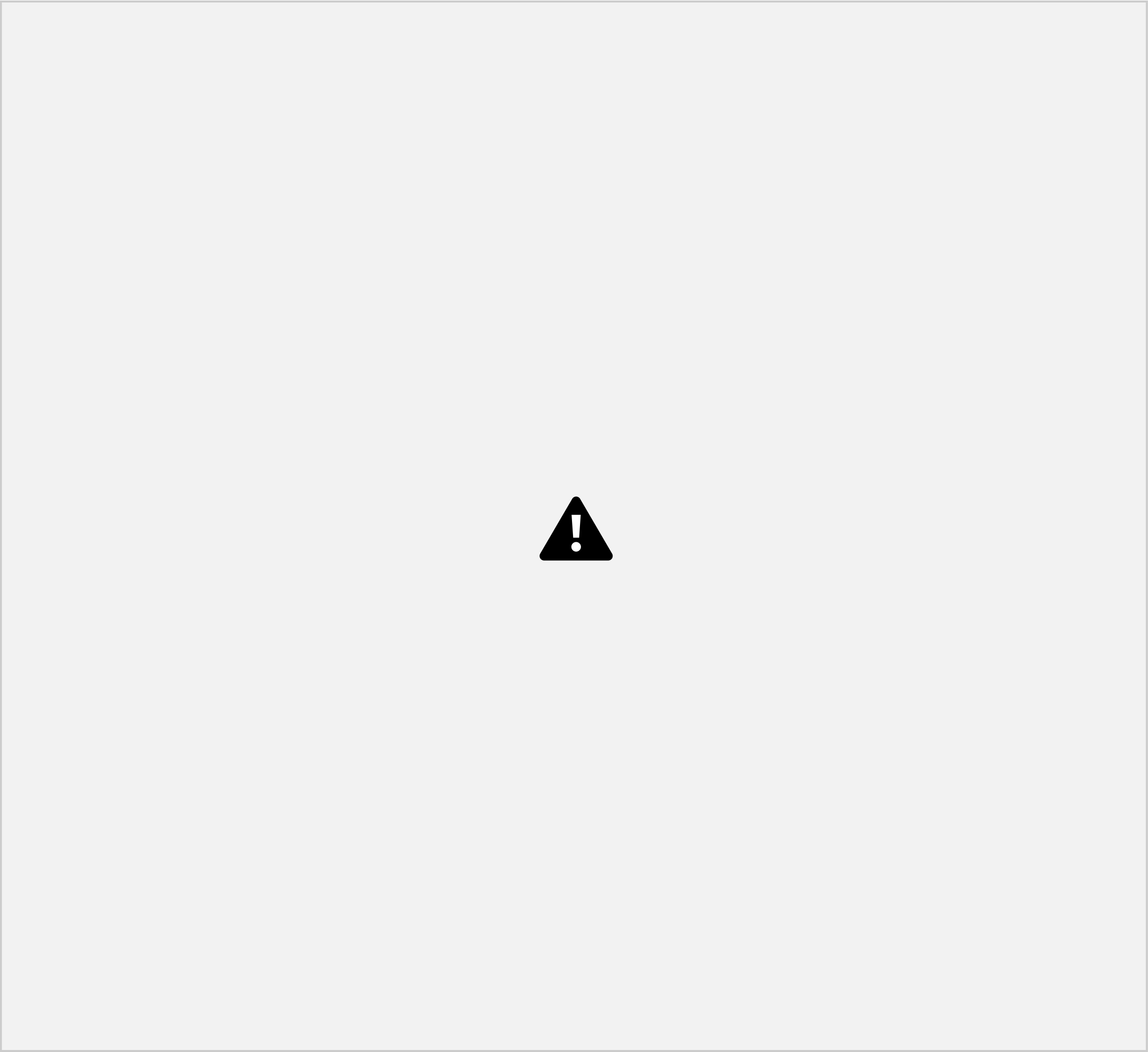
Each array has attributes **ndim** (the number of dimensions), **shape** (the size of each dimension), and **size** (the total size of the array):

NumPy Array Attributes

NumPy Array Attributes

In a one-dimensional array, the ith value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with

Python lists:



NumPy Array Attributes

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

NumPy Array Attributes

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated.

Array Slicing: Accessing Subarrays

TRY IT YOURSELF!!